Document Number: P0247R0
Date: 2016-02-12
Project: ISO SC22/WG21 C++ Standard, Evolution Working Group
Reply to: Nathan Myers nmyers12@bloomberg.net

Criteria for Contract Support

1. Features identified as essential for runtime contract support:

- Check expressions may be placed as assertions in function bodies.

- Check expressions may be annotated to identify those that must not normally be evaluated alongside regular runtime checks, because they could violate function-contract obligations (particularly performance, but also throwing, allocation, etc.).

```
auto binary_search(RAIterator b, RAIterator e, Ordered v) -> bool {
    [[ assert: b <= e ]];                              // regular
    [[ assert audit: std::is_partitioned(b, e,
        [&v](auto const& v2) { return v2 < v; }) ]]; // audit check
    ...
}
```

  [The "audit" seen here is a placeholder for annotating a check not normally evaluated at runtime.]

- Users may select at build time which, if any, checks are to be evaluated at run time. Checks evaluated may include regular checks only, or both regular and audit checks. E.g.

```
$ cc --check-audit -c bsearch.cc    # check everything
$ cc -c bsearch.cc                  # do not evaluate "audit" checks
$ cc --check-none -O -c bsearch.cc  # check nothing
```

- For implementations that permit linking together TUs built with different selections of which checks to evaluate, the standard must not overconstrain which TU's selection determines which checks must be evaluated at a particular call site.

  [I.e.: Checks that annotate an inline function might be evaluated, or not, according to the selection that was made for the TU they get expanded in. Similarly, checks annotating a function template might be evaluated, or not, according to the selection made for the TU where the relevant instantiation lives. But we should not say so, precisely. Instead, the policy an implementer finds easiest or best favored by customers implicitly conforms. With modules, we should be able to be more precise. Note that there is no ambiguity about what a correct program would do.]

- Users may provide a function to call in response to a violated check expression. If no

such function is provided, the response is to terminate execution as if by abort(). E.g.:

```
$ nm handler.o
00000  t  contract_violation_handler(std::source_location const&)
$ cc -c qualify.o bsearch.o handler.o -o qualify
```

[As precedent, the standard already enables users to change core semantics by linking a distinguished symbol, `::operator new()`.]

[Link-time specification of the violation response allows the linker of a DLL to give the DLL its own response to violations on calls into it. More precisely: If a particular DLL specifies a handler, violations caught in that DLL may invoke that handler, regardless of what the rest of the program does. Otherwise, if a user specifies a handler for the main program, violations evoke that handler. Otherwise, the response is `abort()`.]

- The violation-handler function is passed a std::source_location argument to identify the calling context (N4529 [`reflection.src_loc`]).

  [N4529 is a TR, and might not be in C++17. If necessary, we must lift `reflection.src_loc` wholesale from the TR. Implementations are urged to make a "best effort" to populate the `source_location` members so as to be useful to customers, but the standard makes few normative requirements. Recommendation is to report the site of the call to the function reporting the violation, not the violated function itself.]

- If a violation-handler function returns, execution resumes after the check. Implementations may have a build mode in which returning is not permitted.

  [As existing code gets instrumented with checks, it will tend to pass through a state where code in production use provokes violations that must be logged so they can be identified and fixed. The overwhelming majority of software development amounts to adding to or changing existing programs, so success of this feature depends utterly on the ability to use annotated libraries with existing programs. When the Standard Library gets so annotated, very few large programs will run to completion with any kind of checking turned on unless a handler can log the event and continue.]

  [Meanwhile, programs proven to not violate preconditions might be made faster by allowing the compiler to assume the preconditions.]

- Response to violations cannot be specified per-check, per-function, or per-TU. Users should not consider contract annotations to be a reliable control-flow mechanism.

  [Per-TU violation response would have many problems. Which TU actually caught the violation? It depends on inlining, template instantiation placement. Per-check or per-function dispatch would make it a control-flow primitive.]

- Check expressions are not part of the function type.

2. Design Notes

- We acknowledge the desire for declaration-level check annotations. We propose syntax for this case, by example, using vector<> member operator[], achieving what `at` was meant for:

  [Thus:

  ```
  reference operator[](size_type pos)
     [[ pre: pos < this->size() ]] ;

  const_reference operator[](size_type pos)
     [[ pre: pos < this->size() ]] const;
  ```

  When a program calling `operator[]` is built with checking turned on, the value `pos` is checked at runtime, and the misuse trapped. When the program is statically analyzed, any calls that can be proven to violate the precondition (or, even, that cannot be proven not to violate it) may be noted. When the program is built with runtime checking off, there is no runtime checking overhead. When operator[] is used in a constexpr context, a violation is ill-formed, regardless of build parameters.]

- An exception (1) thrown from the handler (2) called in response to the violation of (3) a check expression guarding a noexcept function (4) results in a call to terminate(), just like other exceptions thrown from such a function.

  [A straw vote in Kona left little room for change here. In any case the caller of a function declared noexcept cannot be assumed to be exception-safe at the point of the call.]

- Use of attribute syntax does not imply that the feature is optional. (Other syntax would be acceptable, but would take a lot of work.)

  [Aside from simplifying specification, the major benefit of attribute syntax is that it provides backward compatibility: Annotated code can still be compiled with older compilers.]

- It would be hard to justify not including, in the function body, code for checks found in its own declaration. It would be equally hard to justify not running, at the call site, checks seen in the declaration used.

- Inheriting checks on virtual functions, or requiring that checks on base class virtual interfaces match overriders does not work:

  - It is routine for derived-class implementations of a virtual function interface to widen preconditions and narrow postconditions, vs. base class interface requirements. As a consequence, calls through a checked base-class interface often would be unable to fully exercise the derived-class implementation.

[E.g., a base-class facility accepts US-ASCII strings to display. The derived implementation accepts UTF-8.

```
class Display {
  virtual void post_message(std::string_view s)
  [[ pre: is_ascii(s) ]] = 0;
};

class XDisplay : public Display {
  ...
  void post_message(std::string_view s)
    [[ pre: is_utf8(s) ]] override;
};
```

An instance of `XDisplay` may, by Liskov substitution, be passed into any subsystem expecting a Display, and calls there may enforce that messages have only ASCII encoding. A subsystem that takes an `XDisplay` reference may be passed an `XDisplay` or something else it is mixed into, and such a subsystem can generate and post ASCII and UTF-8 messages.]

- Somewhat less commonly, virtual-function implementations narrow requirements vs. the base-class interface, or present more stringent postconditions, such that static checks on the base interface would not sufficiently check input and output.

  [E.g., a base-class facility may accept any file descriptor, while the derived implementation requires that the file descriptor be non-blocking. The derived implementation, then, can provide latency guarantees not promised in the base interface. Clients that don't know about latency can use the derived facility through the base-class interface, ignoring the derived implementation's latency guarantee.]

- Virtual functions are a special case of the more general fact that different points in the code may see different declarations of what is nominally the same function interface. The declaration of a function template specialization or overload can express checks that would not make sense, or would be entirely inexpressible, on the base template.

  ```
  template <typename It, typename T> It find(It b, It e, T const& t);
    // Not much can be said about the general template

  template <typename T> It find(T const* b, T const* e, T const& t)
      [[ pre: b ]] [[ pre: e ]];
      [[ pre: !std::less<T const*>{}(e, b) ]];
    // Given pointers, it is possible to catch certain misuses.]

   template <typename T, typename Container>
  ```

```
struct Wrap {
    Container c;
    T& operator[](size_t pos);  // nothing to say
};

template <typename T, size_t N>
T& Wrap<T (&)[N]>::operator[](size_t pos) [[ pre: pos < N ]];
```

- Evaluating some checks twice seems tolerable and in general not avoidable, but an optimizer could often elide the extras if it mattered.

- It is impossible, in general, to determine whether one set of checks subsumes another. (It is even tricky to say whether checks on two declarations are the same; name binding in the expressions may differ, although we have ODR weasel-wording for that.) Practically, we cannot assume any relationship between checks on declarations of not-precisely-the-same function.

- Check attributes on function pointers (including function pointers as arguments) should be bound (like a storage class) to the object, not the type or the value. Example:

```
void set_callback(bool (*cb)(int a, int b) [[pre: a < b]])
    [[pre: cb != 0]];
```

- Effects on stable ABIs of the semantics proposed are not immediately obvious. We need review by implementers who have ABI constraints.

- Code that would cause a check expression to evaluate to false in a constexpr context should be treated as ill-formed, regardless of build mode. Failing a constexpr check expression should not be taken as substitution failure. Important questions: Must all the check expressions on a constexpr function be, themselves, constexpr? I.e., should having a non-constexpr check expression keep the function it is on from being used in a constexpr? Should a check expression with indeterminate value affect whether a program is well-formed?

- Check expressions, when evaluated, can themselves violate checks on any functions they call. This does not appear to cause any problems.

- Assertions in member-initializer-lists raise awkward questions that may reasonably be considered later. The problem is that to be most useful, assertions should be allowed between subobject initializers, and be allowed use the value of previously initialized subobjects. However, subobjects can be initialized in a different order than they appear lexically. E.g.,

```
struct A {
    B m_b, m_bb, m_bbb;

    template <typename T> B(int i, int j)
```

```
        : m_bbb{}, [[assert m_b.ok()]] m_bb(m_b), m_b(i, j) {}
};
```

- To use check expressions as guarantees to the optimizer (i.e. “an assertion that is not evaluated but would evaluate to false is UB”) would introduce difficult complications. E.g., re-using the previous example:

```
auto binary_search(RAIterator b, RAIterator e, Ordered v) -> bool {
    [[ assert audit: std::is_partitioned(b, e,
        [v](Value v2) { return v2 < v; }) ]];       // A
    [[ assert: b <= e ]];                            // B
    while (b < e) {
        [[assert: *b <= *e || v < *b || *e < v ]]   // C
        ...
    }
}
```

  Above, assume the compiler is configured so that A is assumed, not checked, and that B and C are to be checked at runtime.

  - Check A has defined behavior only if B is true, so, as normally implemented, an optimizer would be justified in eliding B.

  - A clever enough optimizer could deduce that A implies C, and elide that too.

- Interactions with optimization have complicated effects even without treating check expressions as assumed.

  [Forbidding the optimizer from deducing that an evaluated assertion is true seems to have unfortunate consequences. E.g., it seems to forbid

  - any optimization in a check expression
  - eliding code leading to an assertion, based on implications of code that follows it
  - using such implications to optimize that code

  It is hard to know now to know how to express such a prohibition.

  Preconditions are not proof against these concerns; a precondition on an inlined function is equivalent, for this purpose, to an assertion in the caller. Nested inlines model nested local blocks.]